UNIT 2 (Part 1)

Topic from Pressman Book: Design Principle (Page 109), Software Architecture, Component Level Architecture,

System Design

1. Overview
2. Levels of System Design
   - 2.1. External Design
   - 2.2. Internal Design
3. Objectives of System Design
4. General Design Process
5. Types of Software Design
   - 5.1. Structured Design
   - 5.2. Function Oriented Design
   - 5.3. Object Oriented Design
   - 5.4. User Interface Design
6. Design Concepts (Problem Partitioning, Modularity…Coupling, Cohesion etc.)
7. Strategy/Approaches of Design
   - 7.1. Top Down Design
   - 7.2. Bottom Down Design
   - 7.3. Hybrid Design
8. Levels or Phases of Design
   - 8.1. Interface Design
   - 8.2. Architecture Design
   - 8.3. Detailed Design
9. Principles of Software Design
10. Working of Design Modeling (Designing a Model)

Software Design

Software design is the process of defining the architecture, components, interfaces, and other characteristics of a software system to ensure it meets specific functional and non-functional requirements. It is a critical phase in the software development life cycle (SDLC) that translates the requirements gathered during the requirements analysis phase into a blueprint for constructing the software.

- **Design is a meaningful representation of something to be built.** It can be traced to customer requirements and at the same time assessed for quality against a set of predefined criteria of good design.
- In the **context of software Engineering. Design focuses on four major areas of concern- data, architecture, interface, and components.**

- Software design is the **activity of specifying the nature and composition of a software system satisfying client needs and desires, subject to design constraints.**

- Software design is a mechanism to transform user requirements into some suitable form, which helps the programmer in software coding and implementation. It deals with representing the client's requirement, as described in SRS (Software Requirement Specification) document, into a form, i.e., easily implementable using programming language.

Software Design Involved:

1. **Architecture Design**:
    - This involves the high-level structuring of the software system. It defines the system's overall structure, how different components interact with each other, the data flow, and the communication protocols between components. Architecture design considers the system's scalability, security, performance, and other non-functional requirements.
    - Common software architectures include **layered architecture**, **client-server architecture**, **microservices architecture**, and **service-oriented architecture (SOA)**.
2. **Component Design**:
    - Component design focuses on designing individual parts or modules of the software. Each component is designed to fulfill a specific functionality or set of functionalities and is often implemented as a self-contained unit that can be developed and tested independently.
    - This phase involves deciding how data is managed within each component, the methods and procedures required, and the interfaces for interaction with other components.
3. **Interface Design**:
    - Interface design defines the way components, users, and other systems interact with the software. This includes the design of user interfaces (UI) and application programming interfaces (APIs).
    - UI design focuses on creating a user-friendly experience, considering layout, navigation, accessibility, and responsiveness.
    - API design involves specifying how different components communicate, the methods exposed by each component, and the data formats used for communication.
    -
4. **Data Design**:

    - Data design defines how data is stored, accessed, and managed within the software system. It includes the design of databases, data models, data structures, and data flow within and across different components.
    - The goal is to ensure data integrity, consistency, and efficiency in data handling.
5. **Algorithm Design**:
    - Algorithm design involves creating algorithms that solve specific problems or perform specific tasks within the software. This includes designing the logic, procedures, and rules needed to manipulate data or perform computations.
6. **Security Design**:
    - Security design focuses on protecting the software system and its data from unauthorized access, threats, and vulnerabilities. It includes defining security requirements, access control, encryption, authentication, and authorization mechanisms.
7. **Error Handling and Exception Management**:
    - This involves designing how the software will handle errors and exceptions to ensure the system remains stable and continues to function correctly in the face of unexpected events or inputs.
8. **Design Documentation**:
    - Software design is typically accompanied by detailed documentation that serves as a blueprint for the development team. This includes design specifications, diagrams, flowcharts, pseudocode, and descriptions of the system's architecture and components.

Types of Software Design:

1. **High-Level Design (HLD)**:
    - Also known as **system design** or architecture design, HLD provides an overview of the system, including its architecture, components, modules, and their relationships. It identifies the main building blocks and the data flow between them, serving as a guide for the overall structure of the system.
2. **Low-Level Design (LLD)**:
    - LLD, also known as **detailed design**, focuses on the specifics of each component or module. It includes designing algorithms, data structures, interfaces, and the implementation details of each module. LLD provides a detailed blueprint for developers to write the actual code.

Objectives of Software Design:

- **Meet Requirements**: Ensure that the software meets all functional and non-functional requirements specified by stakeholders.
- **Modularity**: Break down the system into smaller, manageable, and reusable components or modules.
- **Maintainability**: Make the software easy to maintain, extend, and modify over time.
- **Performance**: Optimize the software for performance, ensuring it is efficient and meets performance requirements.
- **Scalability**: Design the software to handle increased loads or expansion with minimal changes.
- **Security**: Ensure the software is secure against threats and vulnerabilities.
- **Usability**: Make the software user-friendly and intuitive for the intended users.

Tools and Techniques Used in Software Design:

- **Unified Modeling Language (UML)**: A standard visual language used to model the architecture, behavior, and structure of software systems. UML includes various diagrams like class diagrams, sequence diagrams, activity diagrams, and use case diagrams.
- **Design Patterns**: Reusable solutions to common design problems, such as Singleton, Factory, Observer, and Decorator patterns.
- **Prototyping**: Creating early models or prototypes of the software to validate and refine design decisions.
- **Software Design Tools**: Tools like Microsoft Visio, Lucidchart, Adobe XD, Balsamiq, and more that help in creating diagrams, wireframes, and mockups for software design.

Consideration points of a good design

- Design is an iterative trial-and-error process.
- Design alternatives should be generated and a decision among them made based on design principles.
- Design alternatives and decisions should be recorded.
- Design of even small programs may be a long and difficult task.

Levels of Software Design

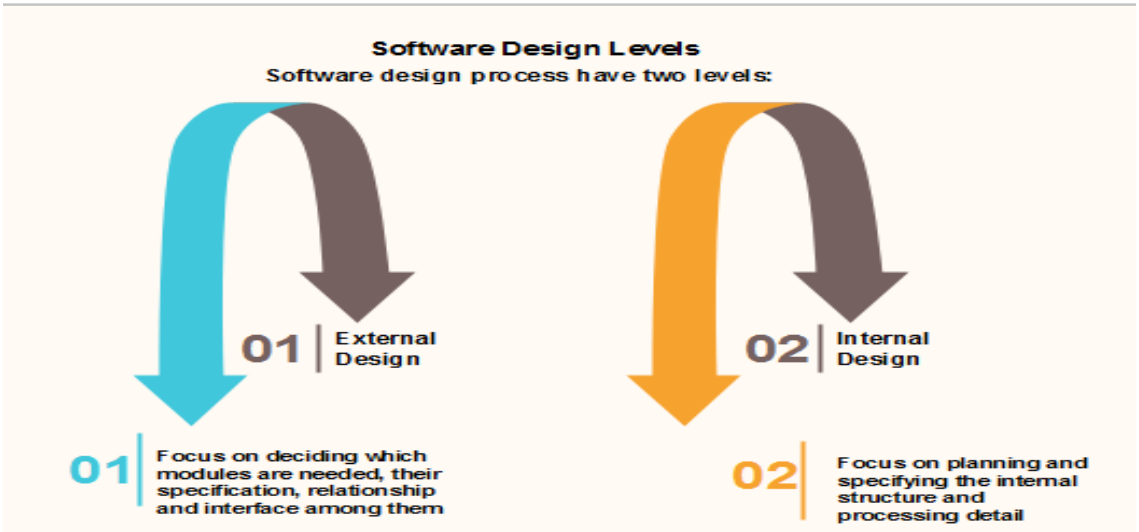
The design process has two levels.

1. First level (External Design)

- Focus is on deciding which modules are needed for the system, the specifications of these modules, and how the modules should be interconnected.
- Known as system design or top-level design.

2. **Second level**

- The **internal design** of the modules, or how the specifications of the module can be satisfied, is decided.
- Known as detailed design or logic design,
- Expands the system design to contain a more detailed description of the processing logic and data structures so that the design is sufficiently complete for coding.



Input & output for design

Inputs:

1. Requirements
2. Environmental Constraints
3. Design Criteria

Output of Design Effort:

1. Architectural design which shows how pieces are interrelated.
2. Specification for any new pieces.
3. Definition for any new data.

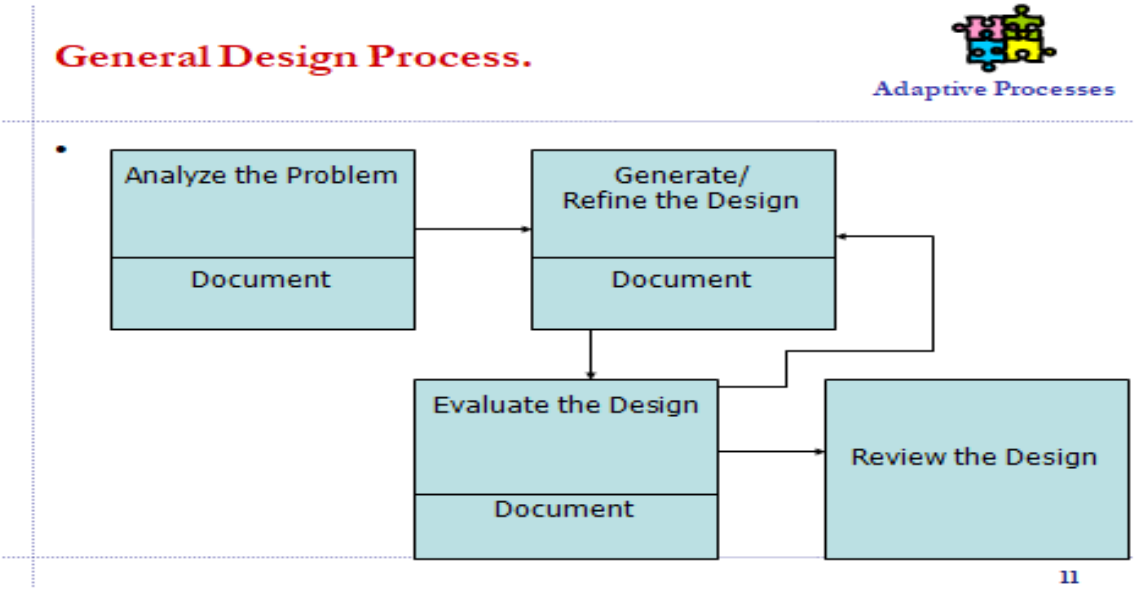
Objectives of Software Design

Following are the purposes of Software design:

Objectives of Software Design

1. **Correctness:**Software design should be correct as per requirement.
2. **Completeness:**The design should have all components like data structures, modules, and external interfaces, etc.
3. **Efficiency:**Resources should be used efficiently by the program.
4. **Flexibility:**Able to modify on changing needs.
5. **Consistency:**There should not be any inconsistency in the design.
6. **Maintainability:** The design should be so simple so that it can be easily maintainable by other designers.

General Design Process



S/w design is a problem solving activity and a general problem solving strategy is comprised of the following steps.

1. Understand the problem.
2. Generate potential solutions to the problem.
3. Evaluate potential solutions to pick the best.
4. If the best solutions are inadequate, return to step 2 to refine solutions or to generate new solutions.
5. Ensure that the solution is adequate, and the work is complete and well documented.

Software Design Concepts

- Problem Partitioning
- Modularity
- Abstraction

Problem Partitioning

For small problem, we can handle the entire problem at once but for the significant problem, divide the problems and conquer the problem it means to divide the problem into smaller pieces so that each piece can be captured separately.

- When solving a small problem, the entire problem can be tackled at once. The complexity of large problems and the limitations of human minds do not allow large problems to be treated as huge monoliths.
- For solving larger problems, the basic principle is the time-tested principle of "divide and conquer. “

"divide into smaller pieces, so that each piece can be conquered separately."

- For software design, partition the problem into sub problems and then try to understand each sub problem and its relationship to other sub problems in an effort to understand the total problem.
- That is goal is to divide the problem into manageably small pieces that can be solved separately, because the cost of solving the entire problem is more than the sum of the cost of solving all the pieces.
- The different pieces cannot be entirely independent of each other, as they together form the system. The different pieces have to cooperate and communicate to solve the larger problem.
- Problem partitioning also aids design verification.

For software design, the goal is to divide the problem into manageable pieces.

Benefits of Problem Partitioning

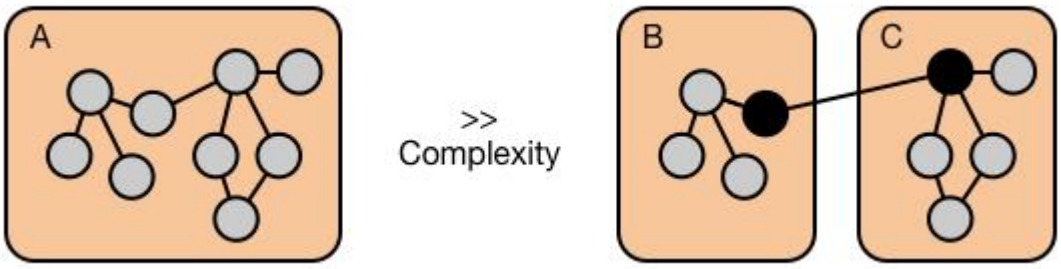
1. Software is easy to understand
2. Software becomes simple
3. Software is easy to test
4. Software is easy to modify
5. Software is easy to maintain
6. Software is easy to expand

These pieces cannot be entirely independent of each other as they together form the system. They have to cooperate and communicate to solve the problem. This communication adds complexity.

Note: As the number of partition increases = Cost of partition and complexity increases

Modularity

Modularity in the software design phase refers to the practice of dividing a software system into distinct, independent modules or components, each encapsulating a specific functionality or a set of related functionalities. This approach aims to create a system that is easier to understand, develop, test, maintain, and extend.



Modularity refers to the compartmentalization and interrelation of the parts of a software package. In software design, modularity refers to a logical partitioning of the "software design" that allows

complex software to be manageable for the purpose of implementation and maintenance.

It is the only property that allows a program to be intellectually manageable. Single large programs are difficult to understand and read due to a large number of reference variables, control paths, global variables, etc.

A system is modular if it is composed of well-defined, conceptually simple and independent units interacting through well-defined interfaces.

Example of module:

Operating System Design

Operating systems (OS) like Linux are highly modular:

- **Kernel Module**: Manages core OS functionalities like memory management, process scheduling, and hardware interfacing.
- **File System Module**: Provides the abstraction for file storage, retrieval, and organization.
- **Network Module**: Manages networking protocols and communications.

- **Device Drivers**: Handle interactions with hardware devices like printers, storage, and graphics cards.
- **User Interface Module**: Manages the graphical or command-line interface for user interaction.

The desirable properties of a modular system are:

- Each module is a well-defined system that can be used with other applications.
- Each module has single specified objectives.
- Modules can be separately compiled and saved in the library.
- Modules should be easier to use than to build.
- Modules are simpler from outside than inside.

Advantages and Disadvantages of Modularity

Advantages of Modularity

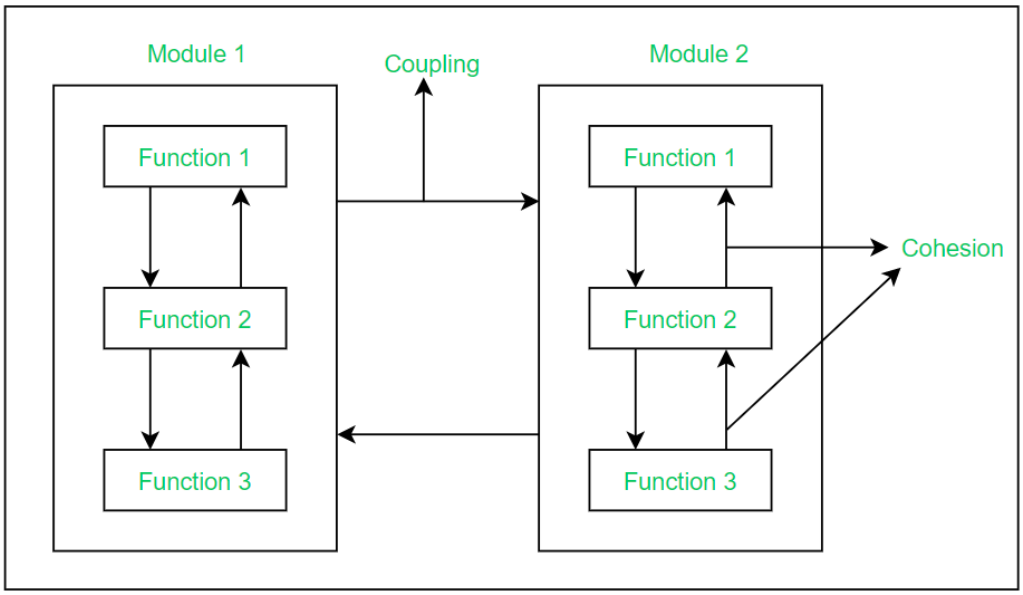
- It allows large programs to be written by several or different people
- It encourages the creation of commonly used routines to be placed in the library and used by other programs.
- It simplifies the overlay procedure of loading a large program into main storage.
- It provides more checkpoints to measure progress.
- It provides a framework for complete testing, more accessible to test
- It produced the well designed and more readable program.

Disadvantages of Modularity

- Execution time maybe, but not certainly, longer
- Storage size perhaps, but is not certainly, increased
- Compilation and loading time may be longer
- Inter-module communication problems may be increased
- More linkage required, run-time may be longer, more source lines must be written, and more documentation has to be done

Key Principal of Modular Design

a) **Functional Independence:** Functional independence is achieved by developing functions that perform only one kind of task and do not excessively interact with other modules. Independence is important because it makes implementation more accessible and faster. The independent modules are easier to maintain, test, and reduce error propagation and can be reused in other programs as well. Thus, functional independence is a good design feature which ensures software quality.



It is measured using two criteria:

- **Cohesion:** Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose. High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes.

- **Coupling:** Coupling refers to the degree of interdependence between software modules. High coupling means that modules are closely connected and changes in one module may affect other modules. Low coupling means that modules are independent and changes in one module have little impact on other modules.

b) Information hiding: The fundamental of Information hiding suggests that modules can be characterized by the design decisions that protect from the others, i.e., In other words, modules should be specified that data include within a module is inaccessible to other modules that do not need for such information.

The use of information hiding as design criteria for modular system provides the most significant benefits when modifications are required during testing's and later during software maintenance. This is because as most data and procedures are hidden from other parts of the software, inadvertent errors introduced during modifications are less likely to propagate to different locations within the software.

Key Characteristics of Information Hiding:

- **Encapsulation**: All internal data and implementation details of a module are encapsulated within the module. The module exposes only the necessary parts (like functions or methods) required for other modules to interact with it.
- **Minimal Exposure**: A module's interface should be minimal and only expose what is necessary for other modules to use its functionality. This reduces the risk of unintended dependencies and keeps the module's internal logic flexible and modifiable.
- **Controlled Access**: Information hiding provides controlled access to a module’s internal data, reducing the likelihood of accidental corruption or misuse of the module’s internal state by other modules.

Abstraction

An abstraction is a tool that enables a designer to consider a component at an abstract level without bothering about the internal details of the implementation. Abstraction can be used for existing element as well as the component being designed.

At the highest level of abstraction– an entire system is a module, whose immediate parts are the sub-systems of which it is composed.
Eg. The immediate parts of a sub-system module– classes, functions, packages or compilation units
The immediate parts of a class module – attributes and operations.

Two common abstraction mechanisms

1. Functional Abstraction
2. Data Abstraction

Functional Abstraction

i. A module is specified by the method it performs.
ii. The details of the algorithm to accomplish the functions are not visible to the user of the function.

Functional abstraction forms the basis for **Function oriented design approaches**.

Data Abstraction

Details of the data elements are not visible to the users of data. Data Abstraction forms the basis for **Object Oriented design approaches**.

Strategy / Approaches of Design

A good system design strategy is to organize the program modules in such a method that are easy to develop and latter too, change. Structured design methods help developers to deal with the size and complexity of programs. Analysts generate instructions for the developers about how code should be composed and how pieces of code should fit together to form a program.

To design a system, there are two possible approaches:

1. Top-down Approach
2. Bottom-up Approach
3. Hybrid

1. Top-down Approach: This approach starts with the identification of the main components and then decomposing them into their more detailed sub-components. We know that a system is composed of more than one sub-systems and it contains a number of components. Further, these sub-systems and components may have their on set of sub-system and components and creates hierarchical structure in the system.

Top-down design takes the whole software system as one entity and then decomposes it to achieve more than one sub-system or component based on some characteristics. Each sub-system or

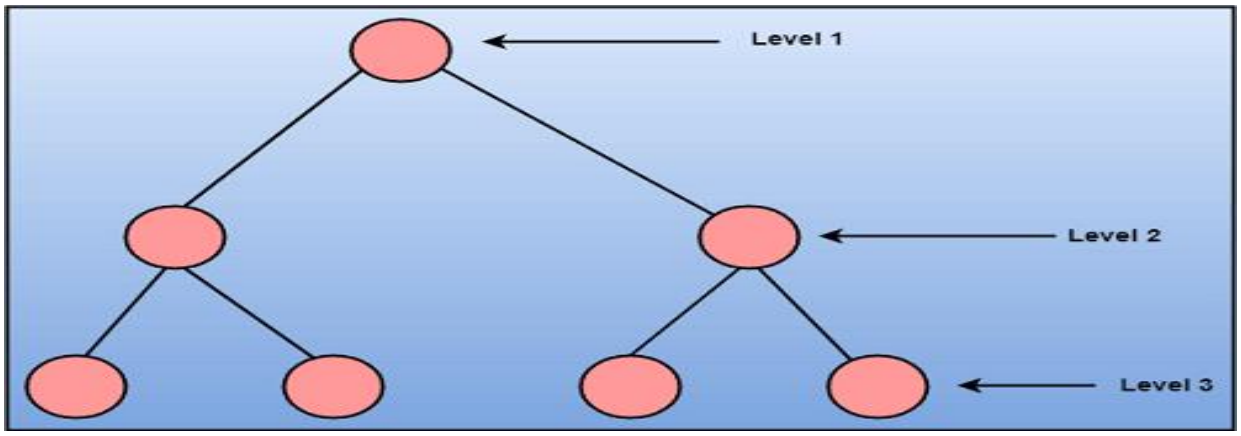
component is then treated as a system and decomposed further. This process keeps on running until the lowest level of system in the top-down hierarchy is achieved.

Top-down design starts with a generalized model of system and keeps on defining the more specific part of it. When all components are composed the whole system comes into existence.

Top-down design is more suitable when the software solution needs to be designed from scratch and specific details are unknown. Top Down is an informal design strategy for **breaking problems into smaller problems**. It has the following **objectives**:-

a. To systematize design process.
b. To produce a modular program design.
c. To provide a **framework** in which problem solving can more efficiently proceed.

A software project is decomposed into sub projects and this procedure is repeated until the subtasks have become so simple that an algorithm can be founded as a solution.



Advantages:

- The main advantage of top down approach is that its strong focus on specific requirements helps to make a design responsive to its requirements.

Disadvantages:

- Its disadvantage is that project and system boundaries tend to be application or specification oriented. Thus it is more likely that advantages of component reuse may be missed.
- The system is likely to miss the benefits of a well structured simple architecture.

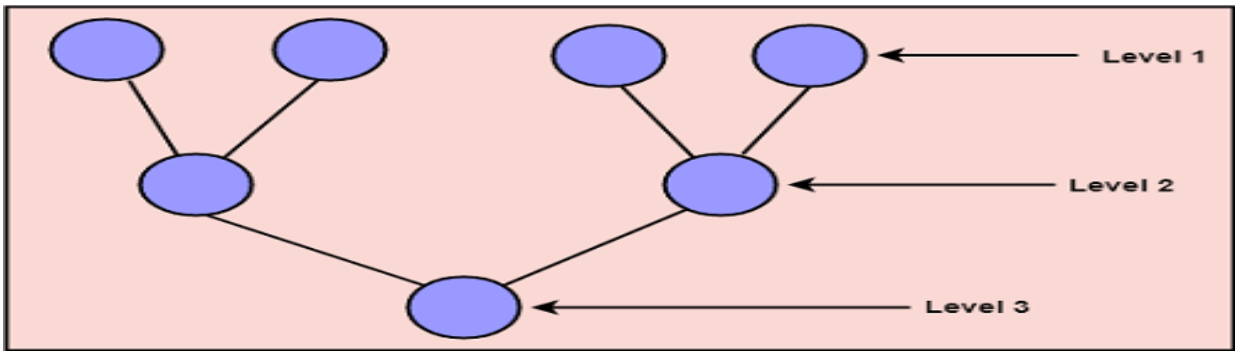
2. Bottom-up Approach: A bottom-up approach begins with the lower details and moves towards up the hierarchy, as shown in fig. This approach is suitable in case of an existing system. The bottom up design model starts with most specific and basic components. It proceeds with composing higher level of components by using basic or lower level components. It keeps creating higher level components until the desired system is not evolved as one single component. With each higher level, the amount of abstraction is increased.

Bottom-up strategy is more suitable when a system needs to be created from some existing system, where the basic primitives can be used in the newer system.

Both, top-down and bottom-up approaches are not practical individually. Instead, a good combination of both is used.

- In case of very complex programs the traditional top down method sometimes breaks down. In its place there has evolved a bottom up style in which **a program is written as a series of layers, each acting as a sort of programming language for the one above**. X Windows is the example of programs written in this style.

- In bottom up design, approach **is to start "at the bottom"** with problems that you already know how to solve (and for which you might already have a reusable software component at hand.) from there you can work upwards towards a solution to the overall problem.
- The code generated using bottom up approach is more maintainable.



Advantages:

- The advantage of bottom up design is the **economies** that result when general solution can be reused.
- It can be **used to hide the low level details of implementations** and be merged with a top down techniques.

Disadvantages:

- It is **not so closely related to the structure of the problem**.
- Its **focus is not on specific requirements** and thus its results may not fit a given need.
- **High quality bottom up solutions prove very hard to construct** and thus most frameworks are to substantial degree under signed.
- It leads to **creation of potential useful functions rather than the most appropriate ones.**

3. Hybrid Design:

It is a combination of both the top-down and bottom-up design strategies. In this, we can reuse the modules.

Consideration for design/ other design Principles

- The design process should not suffer from tunnel vision.
- The design should be traceable to the analysis model.
- The design should not reinvent the wheel.
- The design should minimize the intellectual distance between the software and the problem as it exists in the real world.
- The design should exhibit uniformity and integration.
- The design should be structured to accommodate change.
- The design should be structured to degrade gently, even when aberrant data, events or operating conditions are encountered.
- Design is not coding.

- The design should be assessed for quality as it is being created.
- The design should be reviewed to minimize the conceptual errors.

Coupling and Cohesion

Module Coupling

In software engineering, the coupling is the degree of interdependence between software modules. Two modules that are tightly coupled are strongly dependent on each other. However, two modules that are loosely coupled are not dependent on each other. **Uncoupled modules** have no interdependence at all within them.

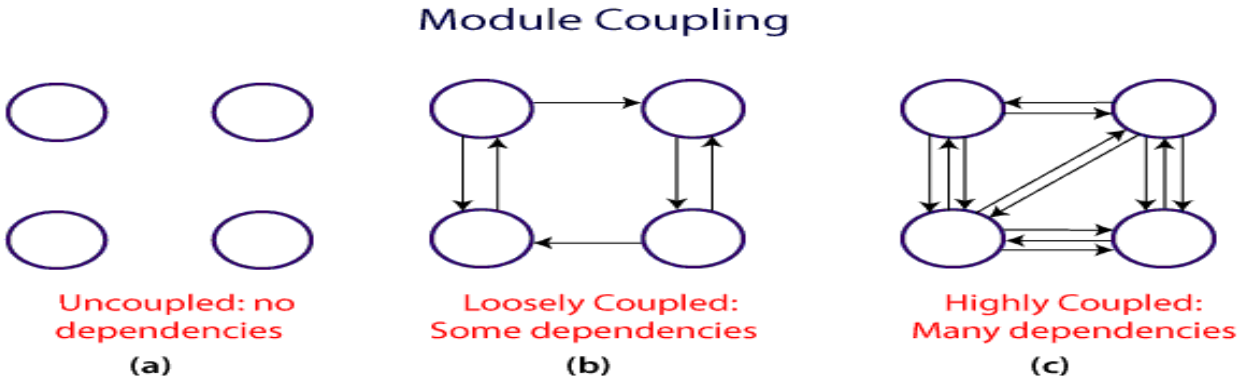
Inter module Coupling:

- ❖ The extent to which modules are independent. Ideally, interdependence amongst modules should be minimized.
- ❖ Coupling is a **measure of the degree of independence between modules**. **Module coupling should be minimized**.
- ❖ Coupling is a property of a collection of modules

Loosely Coupled: When there is little interaction between modules.

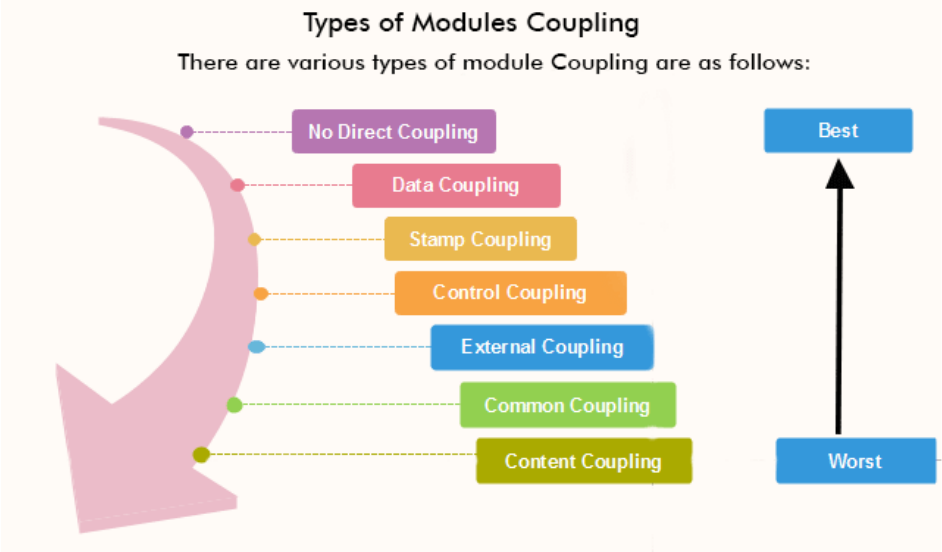
Tightly Coupled: When there is a high degree of interaction between modules.

The various types of coupling techniques are shown in fig:



A good design is the one that has low coupling. Coupling is measured by the number of relations between the modules. That is, the coupling increases as the number of calls between modules increase or the amount of shared data is large. Thus, it can be said that a design with high coupling will have more errors.

Types of Module Coupling



Types of Coupling :

1. **Data Coupling –** Data coupling simply means the coupling of data i.e. interaction between data when they are passed through parameters using or when modules share data through parameters. When data of one module is shared with other modules or passed to other modules, this condition is said to be data coupling.
2. **Control Coupling –** Control coupling simply means to control data sharing between modules. If the modules interact or connects by sharing controlled data, then they are said to be control coupled. The controlled coupling means that one module controls the flow of data or information by other modules by them the information about what to do.
3. **Common Coupling –** Common coupling simply means the sharing of common data or global data between several modules. If two modules share the information through global data items or interact by sharing common data, then they are said to be commonly coupled.
4. **Content Coupling –** Content coupling simply means using of data or control information maintained in other modules by one module. This coupling is also known as pathological coupling. In these coupling, one module relies or depends upon the internal workings of another module. Therefore, if any changes have to be done in the inner working of a module then this will lead to the need for change in the dependent module.
5. **Stamp Coupling –** Stamp coupling occurs in software when two modules communicate by passing a composite data structure (like an object or record), but the receiving module uses only a portion of that structure, leading to a tighter coupling because changes to the overall structure can impact the module even if it doesn't directly use the modified part.
6. **External Coupling –** The external coupling means the sharing of data structure or format that are imposed externally between the modules. External coupling is very important but there should be a limit also. It should be limited to less number of modules with structures.
7. **Message Coupling –** Message coupling means that modules interact with each other by sending or receiving messages instead of directly accessing each other's data or functions. This type of coupling reduces the dependency between modules and makes the system more modular and maintainable.
8. **Data-Content Coupling –** This type of coupling occurs when one module uses data elements or variables directly from another module. It is similar to content coupling but specific to data variables only.
9. **Semantic Coupling –** This type of coupling occurs when two modules are related semantically or logically, and their functions are related to each other. It is a desirable type of coupling as it leads to a better-designed system.
10. **Temporal Coupling –** Temporal coupling occurs when two or more modules must be executed in a specific order or time. For example, a database must be initialized before data can be retrieved from it. This type of coupling should be minimized to reduce system complexity and improve maintainability.
11. **Data-Structure Coupling –** Data-structure coupling occurs when two or more modules share a common data structure, but only part of that structure is used by each module. This type of coupling can lead to unnecessary dependencies and should be avoided if possible.
12. **Functional Coupling –** Functional coupling occurs when two or more modules are tightly coupled due to their shared functionality or task. This type of coupling can be beneficial in certain cases, but it can also lead to a lack of flexibility and increased complexity.

Module Cohesion

In computer programming, cohesion defines to the degree to which the elements of a module belong together. Thus, cohesion measures the strength of relationships between pieces of functionality within a given module. For example, in highly cohesive systems, functionality is strongly related.

Cohesion is an **ordinal** type of measurement and is generally described as "high cohesion" or "low cohesion."

- Cohesion is a property or characteristics of an individual module.
- Cohesion is the measure of strength of the association of elements within a module.
- The extent to which all instructions in a module relate to a single function is called Cohesion.
- In a truly cohesive module, all of the instructions in the module pertain to performing a single unified task is known as Intramodule Cohesion.
- Cohesion in module should be maximized.

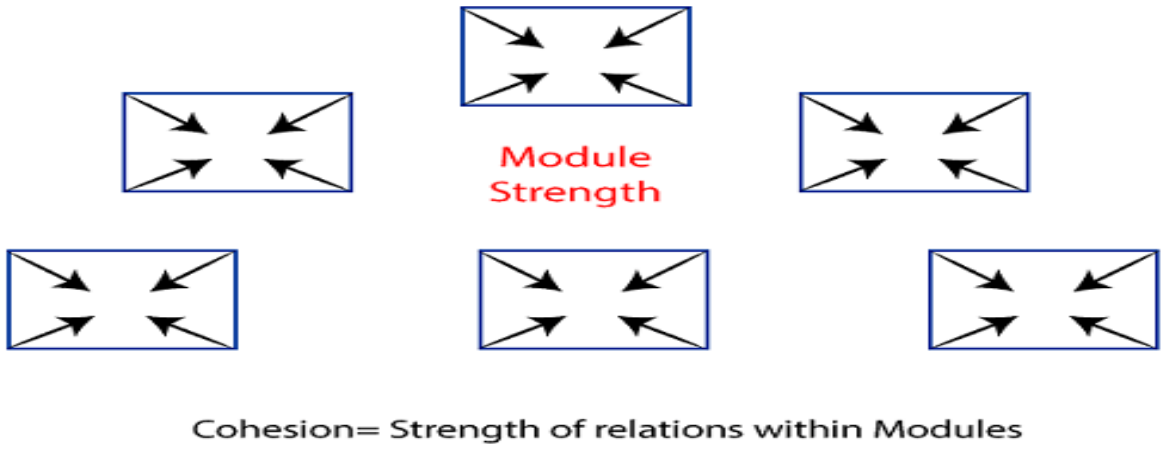
Maximally cohesive modules also tend to be the most loosely coupled, so achieving high levels of cohesion in system design helps minimize coupling.

If a module is designed to perform **one and only one function** then it has **no need to know about the interior workings of other modules**. The cohesive module only needs to take the data it is passed, act on them, and pass its output on to its super-ordinate module.

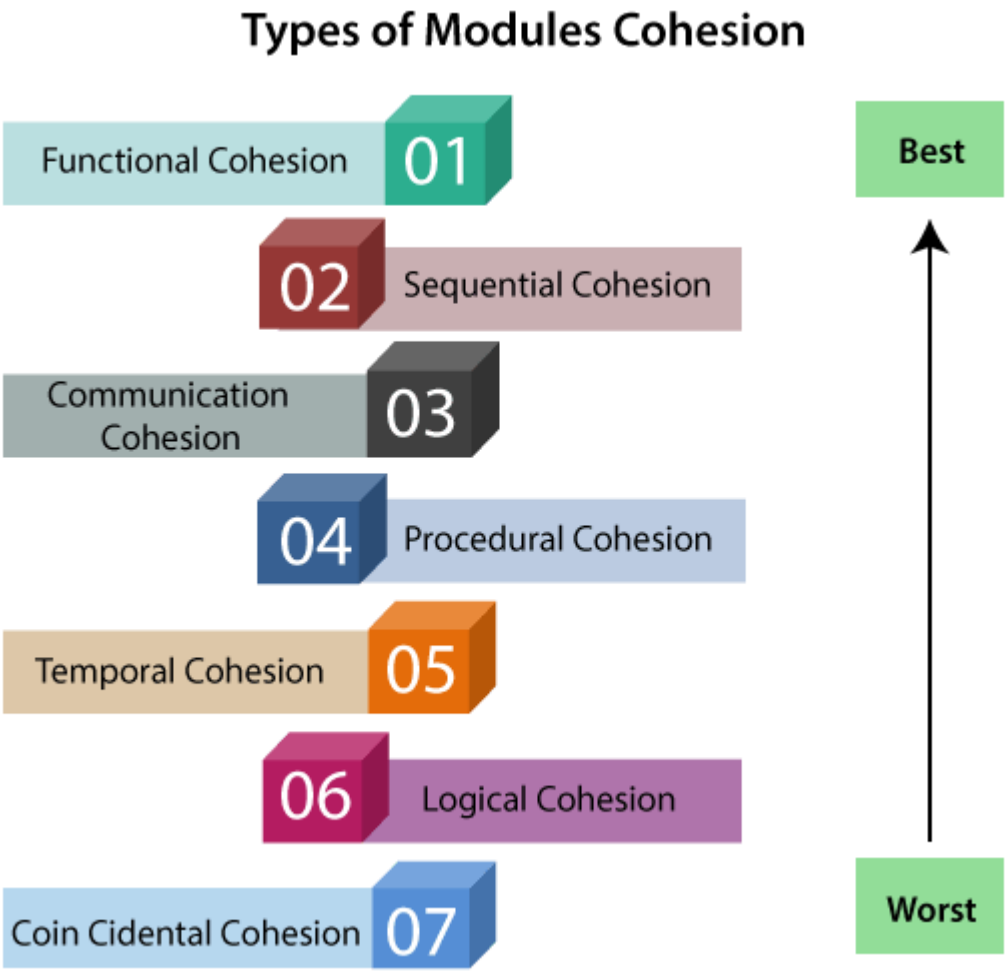
Consideration for Module Cohesion:

-- Cohesion is highest in modules that have a single, clear, logically independent responsibility or role.

-- Cohesion degrades as unrelated responsibilities or tasks are added to a module.



Types of Module Cohesion



Differentiate between Coupling and Cohesion

Coupling	Cohesion
Coupling is also called Inter-Module Binding.	Cohesion is also called Intra-Module Binding.
Coupling shows the relationships between modules.	Cohesion shows the relationship within the module.
Coupling shows the relative **independence** between the modules.	Cohesion shows the module's relative **functional** strength.
While creating, you should aim for low coupling, i.e., dependency among modules should be less.	While creating you should aim for high cohesion, i.e., a cohesive component/ module focuses on a single function (i.e., single-mindedness) with little interaction with other modules of the system.
In coupling, modules are linked to the other modules.	In cohesion, the module focuses on a single thing.

Software Design Methodologies/Techniques/Types

Software design is a process to conceptualize the software requirements into software implementation. Software design takes the user requirements as challenges and tries to find optimum solution. While the software is being conceptualized, a plan is chalked out to find the best possible design for implementing the intended solution.

There are multiple variants of software design. Some of them are as follows:

1. Structured Design

Structured design is a conceptualization of **problem into several well-organized elements of solution.** It is basically concerned with the solution design. Benefit of structured design is, it **gives better understanding of how the problem is being solve**d. Structured design also makes it simpler for designer to concentrate on the problem more accurately.

Structured design is mostly based on 'divide and conquer' strategy where a **problem is broken into several small problems and each small problem is individually solved until the whole problem is solved.**

The small pieces of problem are solved by means of solution modules. Structured design emphasis that these modules be well organized in order to achieve precise solution.

These modules are arranged in hierarchy. They communicate with each other**. A good structured design always follows some rules for communication among multiple modules, namely -**

Cohesion - grouping of all functionally related elements.

Coupling - communication between different modules.

A good structured design has high cohesion and low coupling arrangements.

2. Function Oriented Design

In function-oriented design, **the system is comprised of many smaller sub-systems known as functions. These functions are capable of performing significant task in the system. That is, Function Oriented Design is an approach to software design where the design is decomposed into a set of interacting units where each unit has a clearly defined function.** The system is considered as top view of all functions.

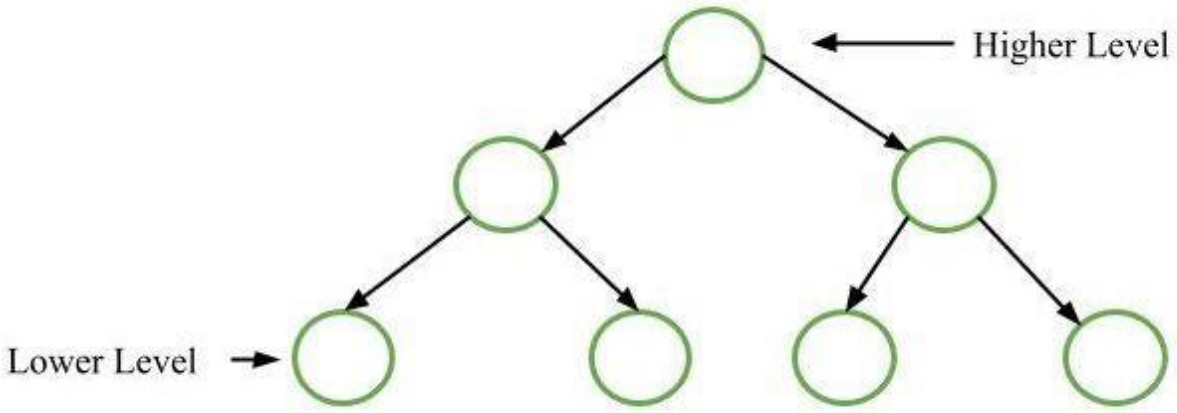
Function oriented design inherits some properties of structured design where divide and conquer methodology is used.

This design mechanism divides the whole system into smaller functions, which provides means of abstraction by concealing the information and their operation. These functional modules can share information among themselves by means of information passing and using information available globally.

Another characteristic of functions is that **when a program calls a function, the function changes the state of the program**, which sometimes is not acceptable by other modules. Function oriented design works well where the system state does not matter and program/functions work on input rather than on a state.

Generic Procedure:

Start with a high level description of what the software / program does. Refine each part of the description one by one by specifying in greater details the functionality of each part. These points lead to Top-Down Structure.



Function Oriented Design Strategies:

Function Oriented Design Strategies are as follows:

Data Flow Diagram (DFD):

A data flow diagram (DFD) maps out the flow of information for any process or system. It uses defined symbols like rectangles, circles and arrows, plus short text labels, to show data inputs, outputs, storage points and the routes between each destination.

Data Dictionaries:

Data dictionaries are simply repositories to store information about all data items defined in DFDs. At the requirement stage, data dictionaries contains data items. Data dictionaries include Name of the item, Aliases (Other names for items), Description / purpose, Related data items, Range of values, Data structure definition / form.

Structure Charts:

It is the hierarchical representation of system which partitions the system into black boxes (functionality is known to users but inner details are unknown). Components are read from top to bottom and left to right. When a module calls another, it views the called module as black box, passing required parameters and receiving results.

Pseudo Code:

Pseudo Code is system description in short English like phrases describing the function. It use keyword and indentation. Pseudo codes are used as replacement for flow charts. It decreases the amount of documentation required.

Design Process for function oriented design

- The whole system is seen as how data flows in the system by means of data flow diagram.
- DFD depicts how functions change data and state of entire system.
- The entire system is logically broken down into smaller units known as functions on the basis of their operation in the system.
- Each function is then described at large.

3. Object Oriented Design

Object oriented design works around the entities and their characteristics instead of functions involved in the software system. This design strategies focuses on entities and its characteristics. The whole concept of software solution revolves around the engaged entities.

Let us see the important concepts of Object Oriented Design:

- **Objects -** All entities involved in the solution design are known as objects. For example, person, banks, company and customers are treated as objects. Every entity has some attributes associated to it and has some methods to perform on the attributes.
- **Classes -** A class is a generalized description of an object. An object is an instance of a class. Class defines all the attributes, which an object can have and methods, which defines the functionality of the object.

  In the solution design, attributes are stored as variables and functionalities are defined by means of methods or procedures.

- **Encapsulation -** In OOD, the attributes (data variables) and methods (operation on the data) are **bundled together is called encapsulation**. Encapsulation not only bundles important information of an object together, but also restricts access of the data and methods from the outside world. This is called information hiding.
- **Inheritance -** OOD allows similar classes to stack up in hierarchical manner where the lower or sub-classes can import, implement and re-use allowed variables and methods from their immediate super classes. This property of OOD is known as inheritance. This makes it easier to define specific class and to create generalized classes from specific ones.
- **Polymorphism -** OOD languages provide a mechanism where methods performing similar tasks but vary in arguments, can be assigned same name. This is called polymorphism, which allows a single interface performing tasks for different types. Depending upon how the function is invoked, respective portion of the code gets executed.

Software Design Process for Objected oriented design

Software design process can be perceived as series of well-defined steps. Though it varies according to design approach (function oriented or object oriented, yet It may have the following steps involved:

- A solution design is created from requirement or previous used system and/or system sequence diagram.
- Objects are identified and grouped into classes on behalf of similarity in attribute characteristics.
- Class hierarchy and relation among them is defined.
- Application framework is defined.

4. User Interface Design

The visual part of a computer application or operating system through which a client interacts with a computer or software. It determines how commands are given to the computer or the program and how data is displayed on the screen.

Types of User Interface

There are two main types of User Interface:

- Text-Based User Interface or Command Line Interface
- Graphical User Interface (GUI)

Text-Based User Interface:
This method relies primarily on the keyboard. A typical example of this is UNIX.

Advantages

- Many and easier to customizations options.
- Typically capable of more important tasks.

Disadvantages

- Relies heavily on recall rather than recognition.
- Navigation is often more difficult.

Graphical User Interface (GUI):

GUI relies much more heavily on the mouse. A typical example of this type of interface is any versions of the Windows operating systems.

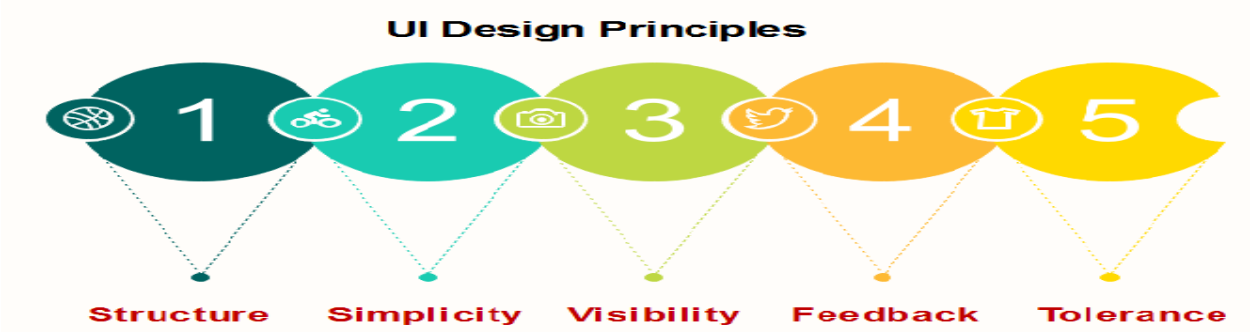
Advantages

- Less expert knowledge is required to use it.
- Easier to Navigate and can look through folders quickly in a guess and check manner.
- The user may switch quickly from one task to another and can interact with several different applications.

Disadvantages

- Typically decreased options.
- Usually less customizable. Not easy to use one button for tons of different variations.

User Interface Design Principles



Structure: Design should organize the user interface purposefully, in the meaningful and usual based on precise, consistent models that are apparent and recognizable to users, putting related things together and separating unrelated things, differentiating dissimilar things and making similar things resemble one another. The structure principle is concerned with overall user interface architecture.

Simplicity: The design should make the simple, common task easy, communicating clearly and directly in the user's language, and providing good shortcuts that are meaningfully related to longer procedures.

Visibility: The design should make all required options and materials for a given function visible without distracting the user with extraneous or redundant data.

Feedback: The design should keep users informed of actions or interpretation, changes of state or condition, and bugs or exceptions that are relevant and of interest to the user through clear, concise, and unambiguous language familiar to users.

Tolerance: The design should be flexible and tolerant, decreasing the cost of errors and misuse by allowing undoing and redoing while also preventing bugs wherever possible by tolerating varied inputs and sequences and by interpreting all reasonable actions.

Good Design Vs Bad Design

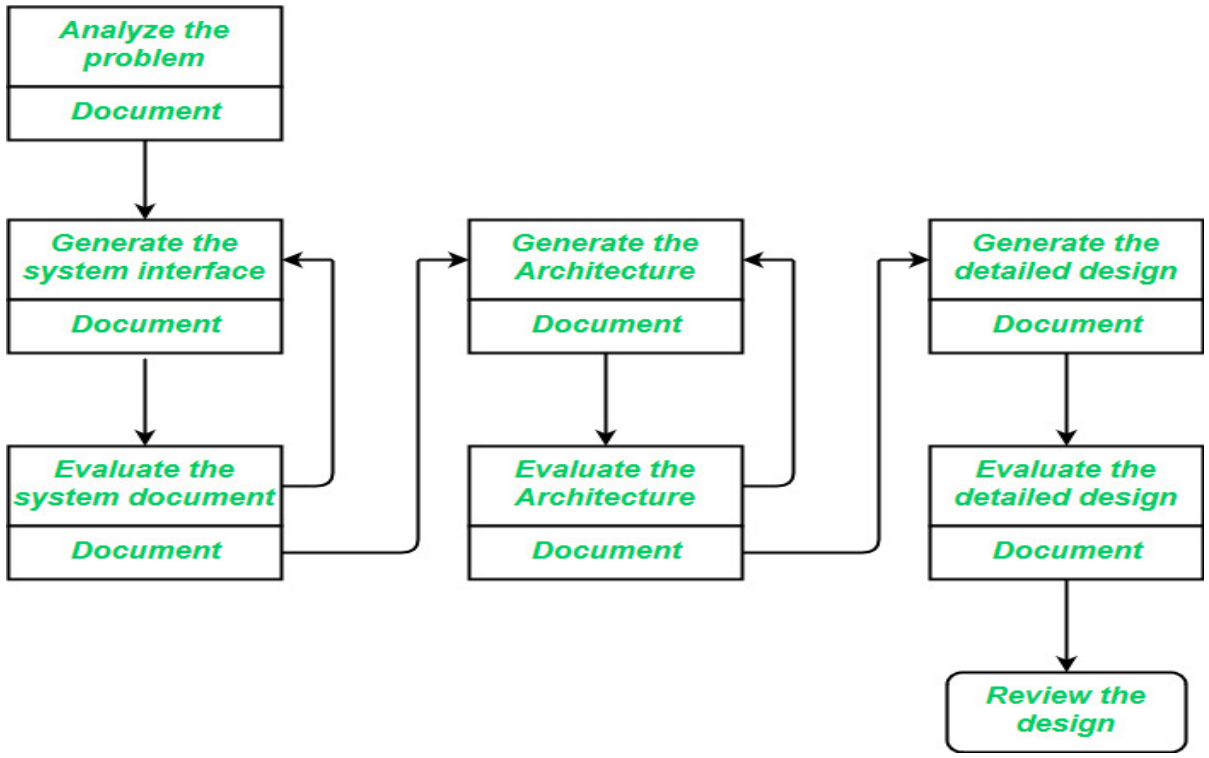
Characteristic	**Good Design**	**Bad Design**
Change	Change in one part does not require change in other part of the system	One conceptual change requires changes to many parts of the system.
Logic	Every piece of logic has one home.	Logic has to be duplicated.
Nature	Simple	Complex
Cost	Small	Very High
Extension	System can be extended with changes in one place	System can’t be extended easily.
Link	The logic link can easily be found	The logic link can’t be remembered.

Levels or Phases of Design (Design Process)

The design phase of software development deals with transforming the customer requirements as described in the SRS documents into a form implementable using a programming language.

The software design process can be divided into the following three levels or phases of design:

1. Interface Design
2. Architectural Design
3. Detailed Design



Interface Design:

Interface design is the specification of the interaction between a system and its environment. This phase proceeds at a high level of abstraction with respect to the inner workings of the system i.e, during interface design, the internal of the systems are completely ignored and the system is treated as a black box. Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts. The design problem statement produced during the problem analysis step should identify the people, other systems, and devices which are collectively called agents.

Interface design should include the following details:

- Precise description of events in the environment, or messages from agents to which the system must respond.
- Precise description of the events or messages that the system must produce.
- Specification on the data, and the formats of the data coming into and going out of the system.
- Specification of the ordering and timing relationships between incoming events or messages, and outgoing events or outputs.

Architectural Design:

Architectural design is the specification of the major components of a system, their responsibilities, properties, interfaces, and the relationships and interactions between them. In architectural design, the overall structure of the system is chosen, but the internal details of major components are ignored.

Architectural design includes:

- Gross decomposition of the systems into major components.
- Allocation of functional responsibilities to components.
- Component Interfaces
- Component scaling and performance properties, resource consumption properties, reliability properties, and so forth.
- Communication and interaction between components.

The architectural design adds important details ignored during the interface design. Design of the internals of the major components is ignored until the last phase of the design.

Detailed Design:

Design is the specification of the internal elements of all major system components, their properties, relationships, processing, and often their algorithms and the data structures.

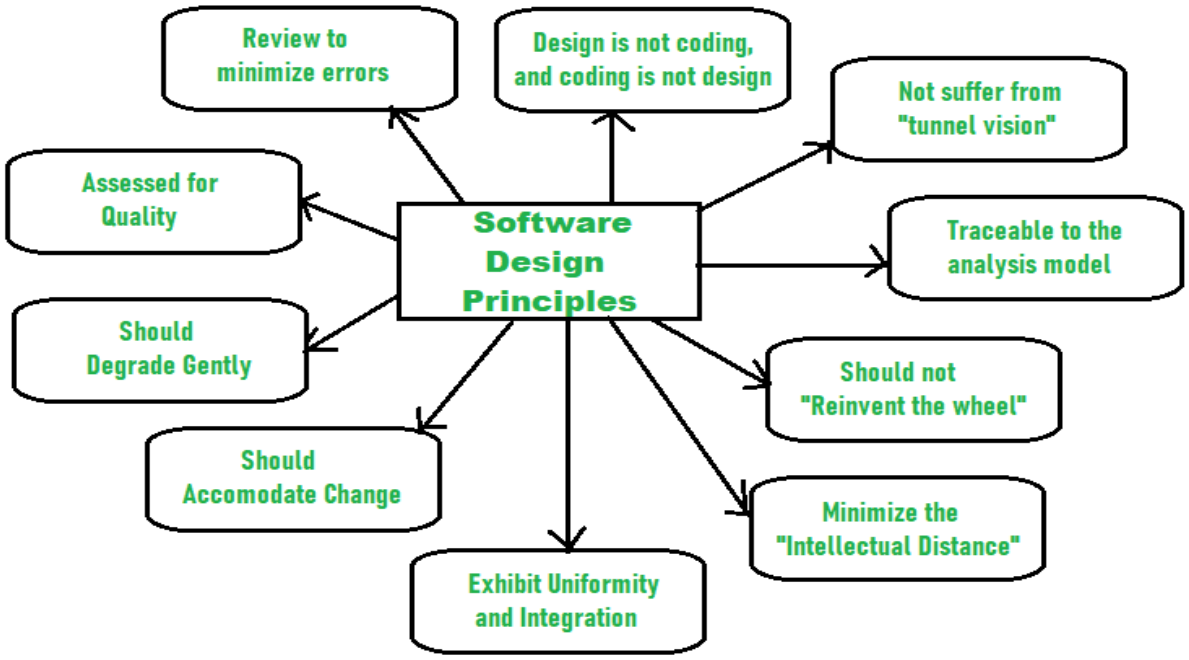
The detailed design may include:

- Decomposition of major system components into program units.
- Allocation of functional responsibilities to units.
- User interfaces
- Unit states and state changes
- Data and control interaction between units
- Data packaging and implementation, including issues of scope and visibility of program elements
- Algorithms and data structures

Principles of Software Design

Design means to draw or plan something to show the look, functions and working of it. **Software Design** is also a process to plan or convert the software requirements into a step that are needed to be carried out to develop a software system. There are several principles that are used to organize and arrange the structural components of Software design. Software Designs in which these principles are applied affect the content and the working process of the software from the beginning.

These principles are stated below:



Should not suffer from "Tunnel Vision" –
While designing the process, it should not suffer from "tunnel vision" which means that is should not only focus on completing or achieving the aim but on other effects also.

Traceable to analysis model –
The design process should be traceable to the analysis model which means it should satisfy all the requirements that software requires to develop a high-quality product.

Should not "Reinvent the Wheel" –
The design process should not reinvent the wheel that means it should not waste time or effort in creating things that already exist. Due to this, the overall development will get increased.

Minimize Intellectual distance –
The design process should reduce the gap between real-world problems and software solutions for that problem meaning it should simply minimize intellectual distance.

Exhibit uniformity and integration –
The design should display uniformity which means it should be uniform throughout the process without any change. Integration means it should mix or combine all parts of software i.e. subsystems into one system.

Accommodate change –
The software should be designed in such a way that it accommodates the change implying that the software should adjust to the change that is required to be done as per the user's need.

Degrade gently –
The software should be designed in such a way that it degrades gracefully which means it should work properly even if an error occurs during the execution.

Assessed or quality –
The design should be assessed or evaluated for the quality meaning that during the evaluation, the quality of the design needs to be checked and focused on.

Review to discover errors –
The design should be reviewed which means that the overall evaluation should be done to check if there is any error present or if it can be minimized.

Design is not coding and coding is not design –
Design means describing the logic of the program to solve any problem and coding is a type of language that is used for the implementation of a design.

Design Modeling in Software Engineering

Design modeling in software engineering represents the features of the software that helps engineer to develop it effectively, the architecture, the user interface, and the component level detail. Design modeling provides a variety of different views of the system like architecture plan for home or building.

Different methods like data-driven, pattern-driven, or object-oriented methods are used for constructing the design model. All these methods use set of design principles for designing a model.

Principles of Design Model

- **Design must be traceable to the analysis model:**

Analysis model represents the information, functions, and behavior of the system. Design model translates all these things into architecture – a set of subsystems that implement major functions and a set of component kevel design that are the realization of Analysis classes. This implies that design model must be traceable to the analysis model.

- **Always consider architecture of the system to be built:**

Software architecture is the skeleton of the system to be built. It affects interfaces, data structures, behavior, program control flow, the manner in which testing is conducted, maintainability of the resultant system, and much more.

- **Focus on the design of the data:**

Data design encompasses the manner in which the data objects are realized within the design. It helps to simplify the program flow, makes the design and implementation of the software components easier, and makes overall processing more efficient.

- **User interfaces should consider the user first:**

The user interface is the main thing of any software. No matter how good its internal functions are or how well designed its architecture is but if the user interface is poor and end-users don’t feel ease to handle the software then it leads to the opinion that the software is bad.

- **Components should be loosely coupled:**

Coupling of different components into one is done in many ways like via a component interface, by messaging, or through global data. As the level of coupling increases, error propagation also increases, and overall maintainability of the software decreases. Therefore, component coupling should be kept as low as possible.

- **Interfaces both user and internal must be designed:**

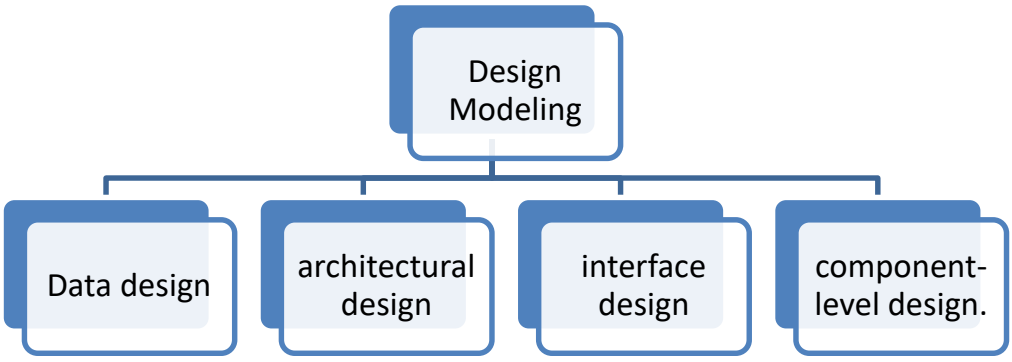
The data flow between components decides the processing efficiency, error flow, and design simplicity. A well-designed interface makes integration easier and tester can validate the component functions more easily.

- **Component level design should exhibit Functional independence:**

It means that functions delivered by component should be cohesive i.e. it should focus on one and only one function or sub-function.

Working of Design Modeling in Software Engineering

Designing a model is an important phase and is a multi-process that represents the data structure, program structure, interface characteristic, and procedural details. It is mainly classified into four categories – Data design, architectural design, interface design, and component-level design.



Data design

It represents the data objects and their interrelationship in an entity-relationship diagram.

Entity-relationship consists of information required for each entity or data objects as well as it shows the relationship between these objects. It shows the structure of the data in terms of the tables. It shows three type of relationship – One to one, one to many, and many to many. In one to one relation, one entity is connected to another entity. In one many relation, one Entity is connected to more than one entity. un many to many relations one entity is connected to more than one entity as well as other entity also connected with first entity using more than one entity.

Data design elements

- The data design element produced a model of data that represent a high level of abstraction.
- This model is then more refined into more implementation specific representation which is processed by the computer based system.
- The **structure of data is the most important part of the software design.**

The data design action translates data defined as part of the analysis model into data structures at the software component level and when necessary into database architecture at the application level.

a) Data Design at the Architectural Level

The challenge in data design is to extract useful information from this data environment.

To solve this challenge**, **data mining techniques, also called knowledge discovery in database (KDD),** is used (that navigate through existing databases in an attempt to extract appropriate business-level information). **An alternative solution, called a data warehouse, adds an additional layer to the data architecture.

b) Data Design at the Component Level

Data design at the component level focuses on the representation of the data structures that are directly accessed by one or more software components.

We consider the following set of principles (adapted from for data specification):

1. *The systematic analysis principles applied to function and behavior should also be applied to data.*
2. *All data structure and the operations to be performed on each should be identified.*
3. *A mechanism for defining the content of each data object should be established and used to define both data and the operation applied it.*
4. *Low-level design decision should be known only to those modules that must make direct use of the data contained within the structure.*
5. *The representation of a data structure should be known only to those modules that must make direct use of the data contained within the structure.*

6. A library of useful data structures and the operations that may be applied to them should be developed.
7. A software design and programming language should support the specification and realization of abstract data types.

Architectural design

The software needs the architectural design to represents the design of software.
IEEE defines architectural design as **"the process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system."**

It defines the relationship between major structural elements of the software.
It is about decomposing the system into interacting components. **It is expressed as a block diagram defining an overview of the system structure – features of the components and how these components communicate with each other to share data.** It defines the structure and properties of the component that are involved in the system and also the inter-relationship among these components.

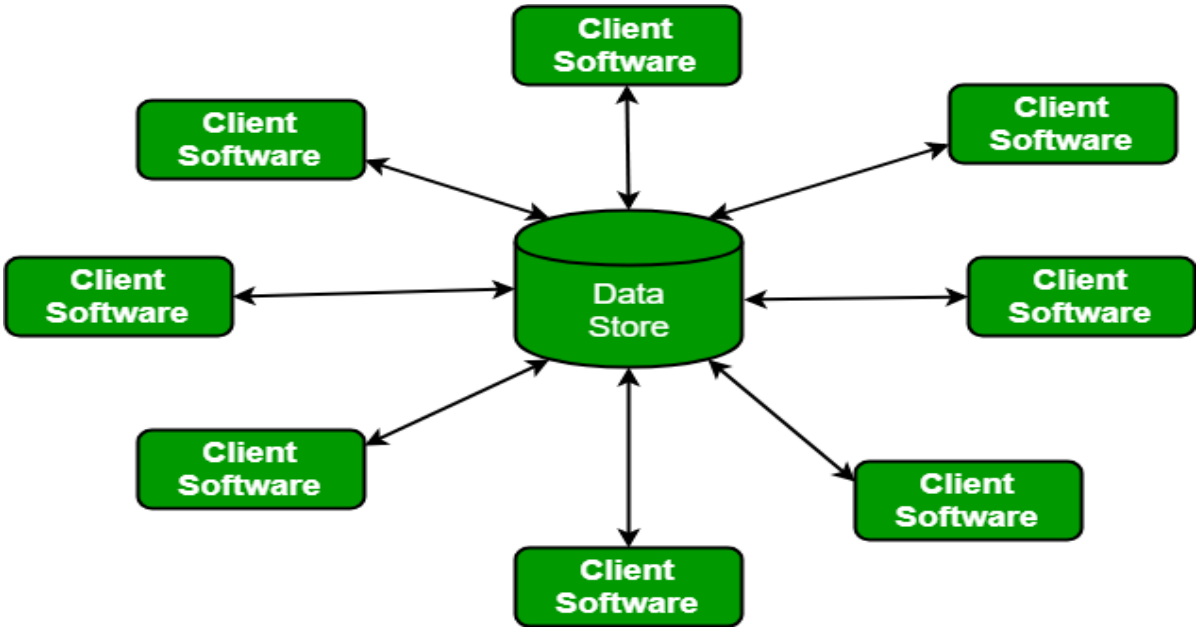
Architectural design elements

- The architecture design elements provide us overall view of the system.
- The architectural design element is generally represented as a set of interconnected subsystem that is derived from analysis packages in the requirement model.
- The architecture model is derived from following sources:
  - The information about the application domain to built the software.
  - Requirement model elements like data flow diagram or analysis classes, relationship and collaboration between them.
  - The architectural style and pattern as per availability.

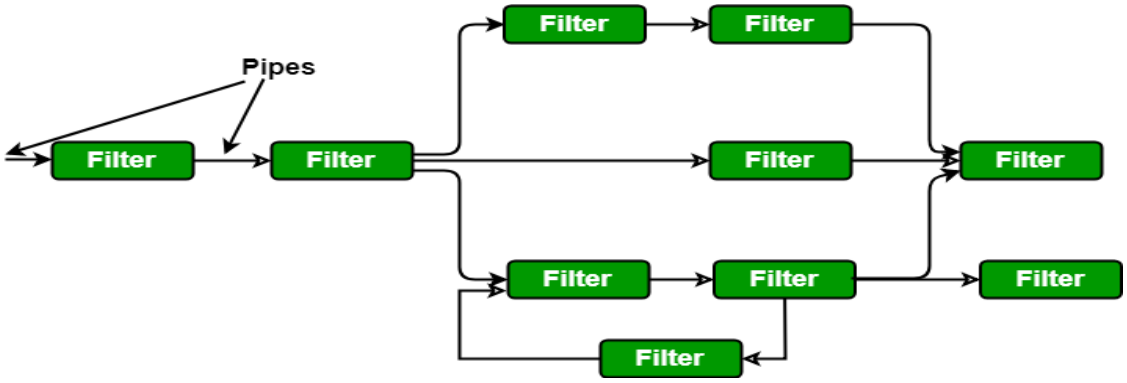
The software that is built for computer-based systems can exhibit one of these many architectural styles. The use of architectural styles is to establish a structure for all the components of the system.

Taxonomy of Architectural styles:

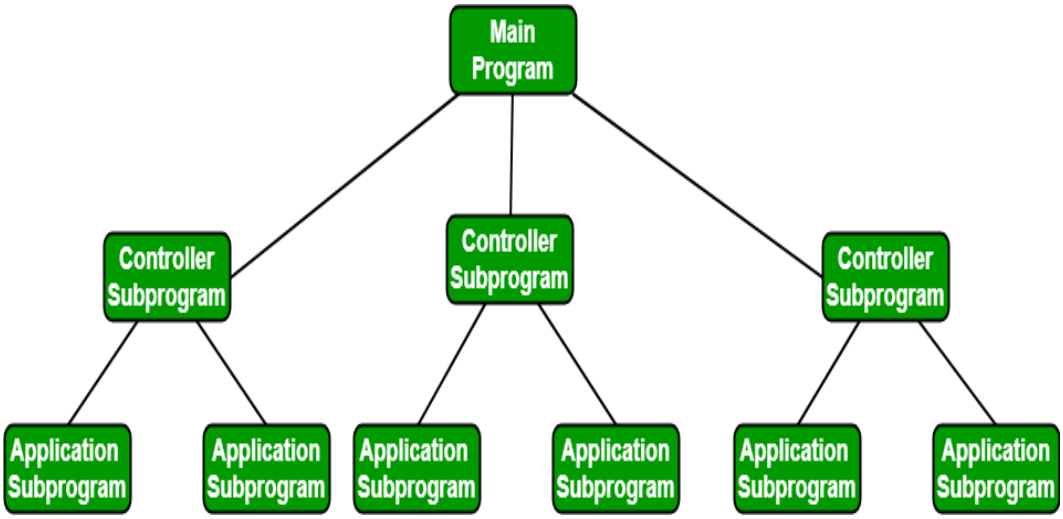
1. **Data centered architectures:**
   - A data store will reside at the center of this architecture and is accessed frequently by the other components that update, add, delete or modify the data present within the store.
   - *The figure illustrates a typical data centered style.* The client software access a central repository. Variation of this approach is used to transform the repository into a blackboard when data related to client or data of interest for the client change the notifications to client software.
   - This data-centered architecture will promote integrability. This means that the existing components can be changed and new client components can be added to the architecture without the permission or concern of other clients.
   - Data can be passed among clients using blackboard mechanism.



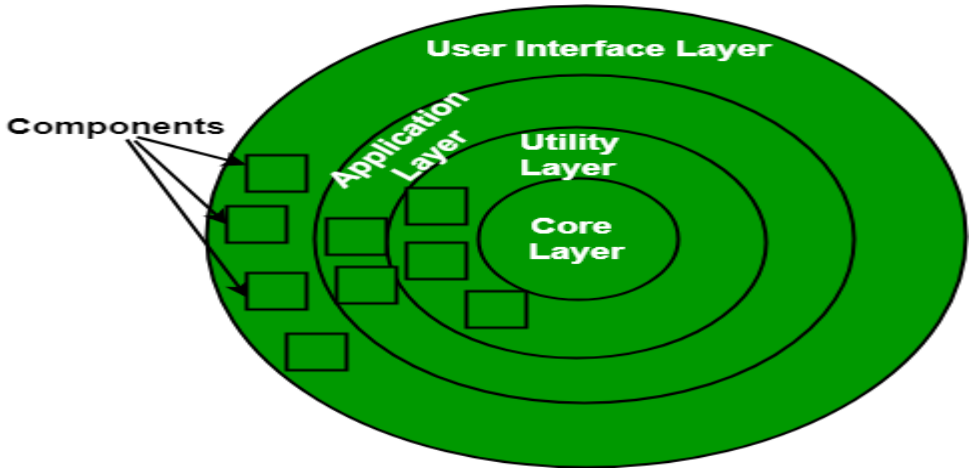
2. **Data flow architectures:**
   - This kind of architecture is used when input data to be transformed into output data through a series of computational manipulative components.
   - **The figure represents pipe-and-filter architecture since it uses both pipe and filter and it has a set of components called filters connected by pipes.**
   - Pipes are used to transmit data from one component to the next.
   - Each filter will work independently and is designed to take data input of a certain form and produces data output to the next filter of a specified form. The filters don't require any knowledge of the working of neighboring filters.
   - If the data flow degenerates into a single line of transforms, then it is termed as batch sequential. This structure accepts the batch of data and then applies a series of sequential components to transform it.



3. **Call and Return architectures:**
   It is used to create a program that is easy to scale and modify. Many sub-styles exist within this category. Two of them are explained below.
   - **Remote procedure call architecture:** This component is used to present in a main program or sub program architecture distributed among multiple computers on a network.
   - **Main program or Subprogram architectures:** The main program structure decomposes into number of subprograms or function into a control hierarchy. Main program contains number of subprograms that can invoke other components.



4. **Object Oriented architecture:**
   The components of a system encapsulate data and the operations that must be applied to manipulate the data. The coordination and communication between the components are established via the message passing.
5. **Layered architecture:**
   - A number of different layers are defined with each layer performing a well-defined set of operations. Each layer will do some operations that become closer to machine instruction set progressively.
   - At the outer layer, components will receive the user interface operations and at the inner layers, components will perform the operating system interfacing (communication and coordination with OS)
   - Intermediate layers to utility services and application software functions.



User Interfaces design

It represents how the Software communicates with the user i.e. the behavior of the system. It refers to the product where user interacts with controls or displays of the product. For example, Military, vehicles, aircraft, audio equipment, computer peripherals are the areas where user interface design is implemented.

Interface design is the specification of the interaction between a system and its environment. This phase proceeds at a high level of abstraction with respect to the inner workings of the system i.e, during interface design, the internal of the systems are completely ignored and the system is treated as a black box. Attention is focused on the dialogue between the target system and the users, devices, and other systems with which it interacts. The design problem statement produced during

the problem analysis step should identify the people, other systems, and devices which are collectively called agents.

Interface design should include the following details:

- Precise description of events in the environment, or messages from agents to which the system must respond.
- Precise description of the events or messages that the system must produce.
- Specification on the data, and the formats of the data coming into and going out of the system.
- Specification of the ordering and timing relationships between incoming events or messages, and outgoing events or outputs.

Interface design elements

1. The user interface
2. The external interface to the other systems, networks etc.
3. The internal interface among various components.

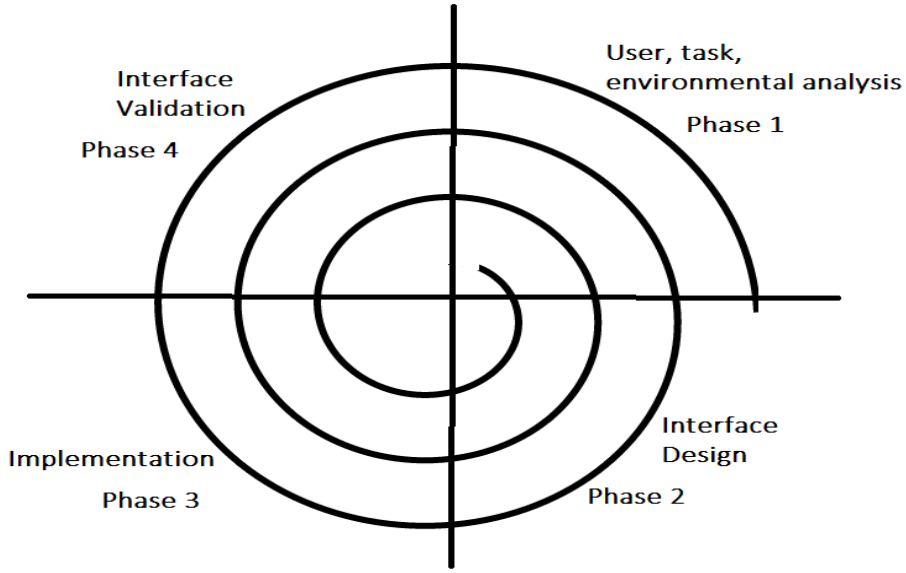
User interface is the front-end application view to which user interacts in order to use the software. The interface design elements for software represent the information flow within it and out of the system. They communicate between the components defined as part of architecture.
The software becomes more popular if its user interface is:

- Attractive
- Simple to use
- Responsive in short time
- Clear to understand
- Consistent on all interface screens

There are two types of User Interface:

1. **Command Line Interface:** Command Line Interface provides a command prompt, where the user types the command and feeds to the system. The user needs to remember the syntax of the command and its use.
2. **Graphical User Interface:** Graphical User Interface provides the simple interactive interface to interact with the system. GUI can be a combination of both hardware and software. Using GUI, user interprets the software.

User Interface Design Process:



The analysis and design process of a user interface is iterative and can be represented by a spiral model. The analysis and design process of user interface consists of four framework activities.

1. **User, task, environmental analysis, and modeling:**

Initially, the focus is based on the profile of users who will interact with the system, i.e. understanding, skill and knowledge, type of user, etc, based on the user's profile users are made into categories. **From each category requirements are gathered. Based on the requirements developer understand how to develop the interface.**

Once all the requirements are gathered a detailed analysis is conducted. In the analysis part, the tasks that the user performs to establish the goals of the system are identified, described and elaborated. The analysis of the user environment focuses on the physical work environment. Among the questions to be asked are:

- Where will the interface be located physically?
- Will the user be sitting, standing, or performing other tasks unrelated to the interface?
- Does the interface hardware accommodate space, light, or noise constraints?
- Are there special human factors considerations driven by environmental factors?

2. **Interface Design:**

The goal of this phase is to define the set of interface objects and actions i.e. Control mechanisms that enable the user to perform desired tasks. Indicate how these control mechanisms affect the system. Specify the action sequence of tasks and subtasks, also called a user scenario. Indicate the state of the system when the user performs a particular task. Always follow the three golden rules stated by Theo Mandel (**Place the user in control, Reduce the user's memory load, Make the interface consistent)**.

Design issues such as response time, command and action structure, error handling, and help facilities are considered as the design model is refined. This phase serves as the foundation for the implementation phase.

3. **Interface construction and implementation:**

The implementation activity begins with the creation of prototype (model) that enables usage scenarios to be evaluated. As iterative design process continues a User Interface toolkit that allows the creation of windows, menus, device interaction, error messages, commands, and many other elements of an interactive environment can be used for completing the construction of an interface.

4. **Interface Validation:**

This phase focuses on testing the interface. The interface should be in such a way that it should be able to perform tasks correctly and it should be able to handle a variety of tasks. It should achieve all the user's requirements. It should be easy to use and easy to learn. Users should accept the interface as a useful one in their work.

Golden Rules for Interface Design:

The following are the golden rules stated by Theo Mandel that must be followed during the design of the interface.

Place the user in control:

- Define the interaction modes in such a way that does not force the user into unnecessary or undesired actions: The user should be able to easily enter and exit the mode with little or no effort.
- Provide for flexible interaction: Different people will use different interaction mechanisms, some might use keyboard commands, some might use mouse, some might use touch screen, etc, Hence all interaction mechanisms should be provided.
- Allow user interaction to be interruptable and undoable: When a user is doing a sequence of actions the user must be able to interrupt the sequence to do some other work without losing the work that had been done. The user should also be able to do undo operation.

- Streamline interaction as skill level advances and allow the interaction to be customized: Advanced or highly skilled user should be provided a chance to customize the interface as user wants which allows different interaction mechanisms so that user doesn't feel bored while using the same interaction mechanism.
- Hide technical internals from casual users: The user should not be aware of the internal technical details of the system. He should interact with the interface just to do his work.
- Design for direct interaction with objects that appear on screen: The user should be able to use the objects and manipulate the objects that are present on the screen to perform a necessary task. By this, the user feels easy to control over the screen.

Reduce the user's memory load:

- Reduce demand on short-term memory: When users are involved in some complex tasks the demand on short-term memory is significant. So the interface should be designed in such a way to reduce the remembering of previously done actions, given inputs and results.
- Establish meaningful defaults: Always initial set of defaults should be provided to the average user, if a user needs to add some new features then he should be able to add the required features.
- Define shortcuts that are intuitive: Mnemonics should be used by the user. Mnemonics means the keyboard shortcuts to do some action on the screen.
- The visual layout of the interface should be based on a real-world metaphor: Anything you represent on a screen if it is a metaphor for real-world entity then users would easily understand.
- Disclose information in a progressive fashion: The interface should be organized hierarchically i.e. on the main screen the information about the task, an object or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

Make the interface consistent:

- Allow the user to put the current task into a meaningful context: Many interfaces have dozens of screens. So it is important to provide indicators consistently so that the user know about the doing work. The user should also know from which page has navigated to the current page and from the current page where can navigate.
- Maintain consistency across a family of applications: The development of some set of applications all should follow and implement the same design, rules so that consistency is maintained among applications.
- If past interactive models have created user expectations do not make changes unless there is a compelling reason.

Component level design

It transforms the structural elements of the software architecture into a procedural description of software components. It is a perfect way to share a large amount of data. Components need not be concerned with how data is managed at a centralized level i.e. components need not worry about issues like backup and security of the data.

Component level diagram elements

- The component level design for software is similar to the set of detailed specification of each room in a house.
- The component level design for the software completely describes the internal details of the each software component.
- The processing of data structure occurs in a component and an interface which allows all the component operations.
- In a context of object-oriented software engineering, a component shown in a UML diagram.

- The UML diagram is used to represent the processing logic.

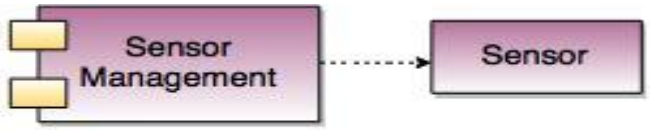
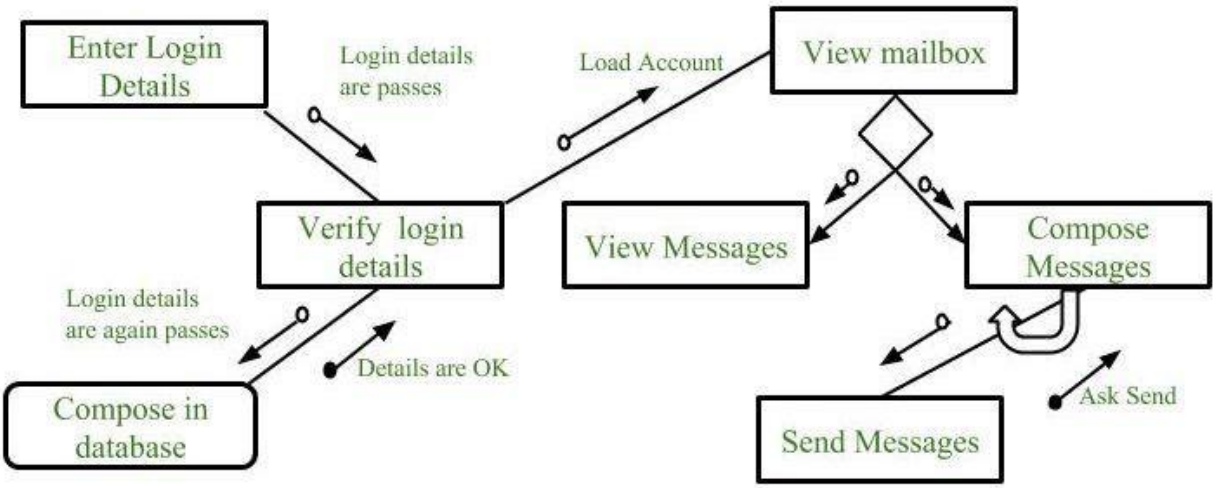


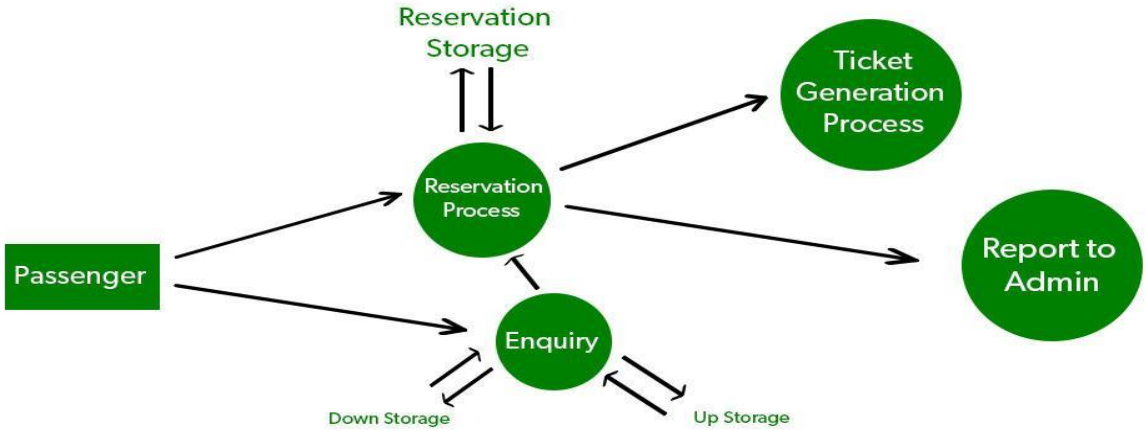
Fig. - UML component diagram for sensor managemnet

Example of Structure Chart

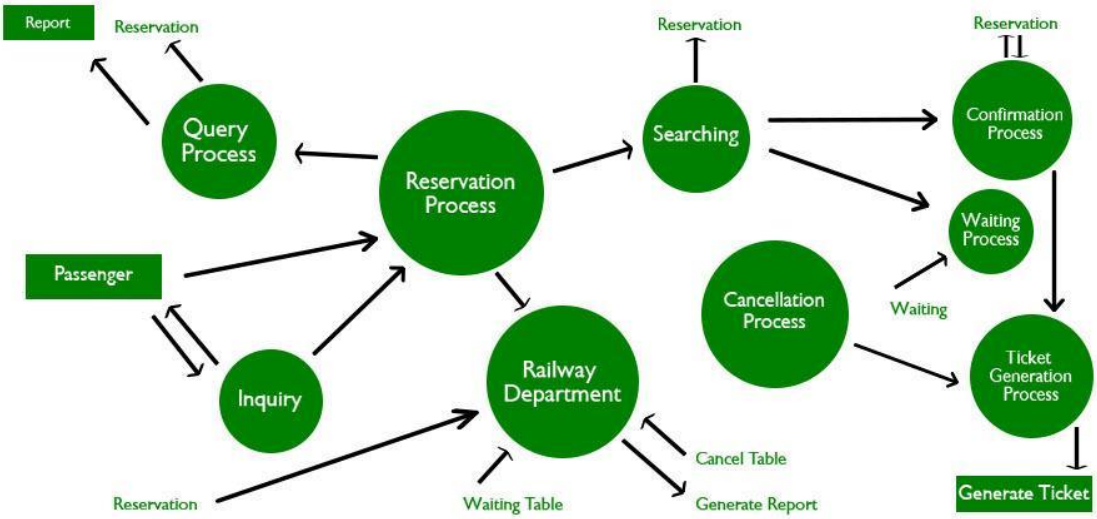


Example of DFD

0-LEVEL DFD



1-LEVEL DFD


Report
Reservation
Query Process
Reservation Process
Passenger
Inquiry
Reservation
Railway Department
Waiting Table
Reservation
Searching
Reservation
Confirmation Process
Waiting Process
Cancellation Process
Waiting
Ticket Generation Process
Cancel Table
Generate Report
Generate Ticket

2-LEVEL DFD